Instruction Set Limitation in Support of Software Diversity

Bjorn De Sutter*, Bertrand Anckaert, Jens Geiregat, Dominique Chanet, and Koen De Bosschere

Ghent University, Electronics and Information Systems Department
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
bjorn.desutter@elis.ugent.be

Abstract. This paper proposes a novel technique, called instruction set limitation, to strengthen the resilience of software diversification against collusion attacks. Such attacks require a tool to match corresponding program fragments in different, diversified program versions. The proposed technique limits the types of instructions occurring in a program to the most frequently occurring types, by replacing the infrequently used types as much as possible by more frequently used ones. As such, this technique, when combined with diversification techniques, reduces the number of easily matched code fragments. The proposed technique is evaluated against a powerful diversification tool for Intel's x86 and an optimized matching process on a number of SPEC 2006 benchmarks.

Key words: diversity, binary rewriting, code fragment matching, software protection

1 Introduction

Collusion attacks on software involve the comparison of multiple versions of an application. For example, an attacker can learn how encryption keys are embedded in an application by comparing two or more versions that embed different keys. Similarly, an attacker can compare two application versions to discover how fingerprints are embedded in them. Or an attacker can compare an application before a security patch has been applied to the same application after the patch has been applied to discover the vulnerability that was addressed by the patch, and then use that information to attack unpatched versions.

Attackers also want to distribute their cracks of popular software. These cracks are nothing more than scripts that automate the attack that was devised manually by the attacker. Generating these scripts is rather easy: it suffices to perform a checksum on the application on which the script will be applied to

* The authors want to thank the Fund for Scientific Research - Flanders (FWO), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), and Ghent University for their support.

make sure that that application is identical to the one originally cracked, and to apply the necessary transformations as simple offset-based binary code patches.

In the above scenarios attackers exploit the valid assumption that different copies of the software are lexically equivalent where they are semantically equivalent. So a natural way to defend against these attacks is to break this assumption. The goal of diversification is therefore to make sure that semantically equivalent code fragments are not lexically equivalent, and that the semantically equivalent, corresponding code fragments constituting two program versions are not easily recognized as being corresponding fragments. If diversification is successful, more effort will be needed to discover true semantical differences between program versions (in the form of keys, fingerprints or patched vulnerabilities), and it will make it much harder to develop an automated script that applies a crack to all versions of some application.

A defender can start from one single master copy of an application, and make diversified copies of the master by applying a unique set of transformations to each copy. These transformations include compiler transformations such as inlining, code factoring, tail duplication, code motion, instruction rescheduling, register reallocation, etc. as well as transformations that have been developed for obfuscating programs, such as control flow flattening, branch functions, and opaque predicates. For each copy, the transformations can be applied at different locations, or with different parameters. While none of these transformations in isolation make it very hard to match corresponding fragments in diversified programs, the combined application of all these techniques does make it harder.

In this paper, we propose a novel transformation for software diversification that will make the matching even harder. *Instruction set limitation* (ISL) replaces infrequently occurring types of instructions in programs with more frequently occurring types of instructions. By itself, this transformation does not make programs more diverse. But by eliminating infrequently occurring instructions, this technique does limit the number of easier targets for a tool that tries to match corresponding fragments in diversified program versions. As such, the proposed technique strengthens the resilience of existing diversification techniques against tools for matching program fragments.

The remainder of this paper is structured as follows. Section 2 provides background information and related work on program fragment matching, diversifying transformations, and instruction selection. The concept of ISL is discussed in Section 3, and an algorithm is proposed in Section 4. ISL is evaluated on native Intel's x86 code in Section 5, and conclusions are drawn in Section 6.

2 Background

In any of the above attacks against diversified software, an attacker first has to try to match corresponding code fragments in different software versions. For any non-trivial program and any non-trivial diversity, automated support is needed in the form of a tool that generates a list of estimated matches between code fragments. The accuracy of such a matcher can be described in terms of false positives and false negatives. These are the fractions of the estimated matches that are not real matches, and the fraction of the real matches that are not included in the estimated matches. Any diversification should try to increase the false positive matching rate and the false negative matching rate. In the remainder of this section, we briefly introduce the inner workings of matchers, of diversification tools, and of the role of instruction selection in these tools.

The attack model we will use here is that of an attacker that can observe a program or a program's execution in every detail. In this malicious host attack model, the attacker has full control over the host machine(s) running the software under attack. These might be real machines, or virtual machines, or a combination of both, that can run, for example, binary instrumentation tools such as valgrind [1] or Diota [2].

2.1 Code Matching

Attackers are most often only interested in understanding or changing the behavior of software. They are not interested in parts of the software that do not contribute to its behavior. Hence attackers are only interested in the code that actually gets executed. This implies that attackers can run a program, observe it, collect data on the executed code, and then use that data in a matching tool. In other words, the matching tool can be guided by dynamic information. A formal description of how to construct matchers using dynamic information is presented in [3]. Here we focus on the fundamental concepts.

Several kinds of information can be used by a matching tool to compare code fragments such as instructions or basic blocks. The *instruction encodings* can be considered, which consist of opcodes and type of operands. Furthermore, the values of *data produced and consumed* by instructions can be used. Or the *execution count*, i.e. the number of times that an instruction is executed for a specific input to the program. An excellent base for comparison is also provided by the first execution count: i.e. the order in which instructions are executed for the first time. And a matcher can consider the locations at which *system calls* occur, together with the arguments passed to the operating system. Using any combination of these types of information, a matcher can assign confidence levels to instruction pairs, indicating with what confidence the matcher believes the pair to be an actual match. The final estimated mapping then consists of all those pairs of which the confidence level surpasses a certain threshold. Obviously, when the threshold is increased, fewer false positives will be found, but this will likely be at the expense of increasing the false negative rate. And vice versa.

None of the above types of matching information take context into account. Instead they describe local properties of single instructions. On top of that, *control flow* and *data flow* information can be considered. Suppose that we already have an estimated mapping based on the local information. By observing a program's execution, an attacker can reconstruct a dynamic control flow graph (including only the executed code and executed control transfers), and a dynamic data dependence graph. The existing mapping can then be refined by taking into account, for each instruction or basic block in these graphs, how well their context matches. For example, consider two instructions in two program versions that the matcher considers as potential matches, albeit at a very low confidence level. The matcher can then take into account these instructions' successors and predecessors in the control flow graph and data flow graph of both program versions. If the successors and predecessors were previously matched with high confidence, the matcher can increase the confidence of the match between the two instructions themselves as well.

There are four mechanisms to combine matchers based on different types of information: combination, limitation, iteration, and bounding. First, matchers can take into account *combinations* of confidence levels, and use combined thresholds instead of thresholds on the individual confidence levels. Secondly, matchers can limit the number of estimated matching pairs per instruction to a certain upper limit. This *limitation* heuristic relies on the assumption that an instruction in one program version usually corresponds to at most a few instructions in another version. Besides resulting in more accurate results, limitation can also speed up the matching because smaller sets of possible matching candidates will be considered. This is particularly the case in *iterative matchers*. In each iteration, an iterative matcher either extends the existing estimated mapping by adding new pairs that surpass its confidence level, or it can filters the existing mapping by throwing out pairs that fall below its confidence threshold. Finally, *bounding* can be used to speed up the matching process, and to consider more contextual information of instructions. With bounding, matchers are first applied to basic blocks instead of instructions. There are much fewer basic blocks, so matching basic blocks will be a faster process. Furthermore, considering a fixed number of successor or predecessor basic blocks in the control flow graphs or data dependency graphs will take into account much more context than considering the same number of successor or predecessor instructions.

During our research on matching and diversification, we developed the matching system described in Table 1. This system was developed and optimized by means of an interactive tool that enables easy exploration of the matching system design space and that shows statistical results on false rates, as well as individual cases of false matches. A more detailed discussion of all the material discussed in this section and the components of our prototype system, including the merits and caveats of different types of matchers, are discussed in detail in [3].

2.2 Diversification

Many program transformations have been developed in compiler research. By applying them selectively in different places, different versions of an application can be generated starting from the master program. To implement this, it suffices to add an additional precondition (i.e., a validity check) for each transformation that is based on two parameters. One of them will be a user-specified probability p, which can be different for each type of transformation, and the other will be a number n generated by a pseudo-random generator. If the generated numbers are in the interval $[0, 1]$ and the added precondition is $n >= p$, then p denotes

Iteration	Granularity	Phase	#Matches	Classifier	Threshold	Direction	Δ
1	bbl	Init	1	Syscalls	0.3		
2	bbl	Extend	1	Encoding	0.5		
				Data	0.5		
				Order	0.5		
				Freq.	0.5		
3	bbl	Extend	2	Encoding	0.1		
				CF	0.1	BOTH	3
4	bbl	Filter		CF	0.1	UP	3
5	bbl	Filter		CF	0.1	DOWN	3
6	bbl	Filter		DF	0.1	UP	3
7	bbl	Filter		DF	0.1	DOWN	3
8	bbl	Extend	2	Data	0.7		
				DF	0.3	BOTH	3
9	bbl	Extend	2	Encoding	0.7		
				CF	0.3	BOTH	3
10	trans	Init	1	Syscalls	0.3		
11	trans	Extend	1	Encoding	0.5		
				Data	0.5		
				Order	0.5		
				Freq.	0.5		
12	trans	Extend	2	Encoding	0.1		
				CF	0.1	BOTH	3
13	trans	Extend	2	DATA	0.7		
				DF	0.3	BOTH	3
14	trans	Extend	2	Encoding	0.7		
				CF	0.3	BOTH	3
15	ins	Extend	3	Encoding	0.6		
				CF	0.1	BOTH	5
16	ins	Filter		CF	0.1	BOTH	5
17	ins	Filter		DF	0.1	BOTH	5
18	ins	Filter		Encoding	0.1		
19	ins	Filter		Data	0.1		
20	ins	Filter		Freq.	0.1		
21	ins	Extend	1	Encoding	0.1		
				CF	0.5	BOTH	5

Table 1. Settings of the default matching system: 21 matchers are applied iteratively, some of which combine difference types of information. The first 9 operate at the basic block level, the next 5 perform the transition from basic blocks to instructions by executing instruction-level matchers bound by the basic block result. Finally, 7 matching steps are performed at the instruction level, not bound by the basic block results. All confidence thresholds are for a confidence scale of $[0, 1]$. For each iteration, the maximum number of selected matches is presented. For the context-aware matchers, the direction is given in which neighboring nodes are traversed, and the distance, i.e. the length of that traversal. UP refers to predecessors, DOWN to successors.

the probability with which a transformation will be applied. The diversity is maximized by maximizing $p(1-p)$, which happens for $p = 0.5$.

In some cases, more than two alternatives (to transform or not to transform) are available. For example, code schedulers can generate many different schedules, much more than two per code fragment, and many different types of opaque predicates can be inserted. In those cases, slightly more complex decision logic needs to be implemented. Still, they all can be normalized to binary choices.

As some transformations involve the insertion of extra code in a program or code duplication, applying all transformations with probability 0.5 may slow down or bloat the program code significantly. To limit the overhead, two approaches can be taken. First of all, smaller values for p can be used. Secondly, the application of the transformations that insert overhead in the program can be limited to certain parts of the program that are determined by profiling the programs. For example, to limit the code size overhead, one can limit the transformations to those code fragments that are executed for most of the common (types of) program input. Or to limit the performance overhead, one can limit the transformations to code that is executed only infrequently.

Finally, one needs to take into account the practicality of selectively applying transformations as a diversification technique. For example, applied transformations should likely survive later transformations, rather than being undone. Furthermore, as recompiling a whole application for each sold copy is not viable, applied transformations should not require an entire recompilation. For these reasons, we believe that the feasible transformations are limited to compiler back-end transformations or to transformations that can be applied in a post-pass program rewriter, such as a link-time rewriter. Our prototype diversifier is based on the x86 backend of the Diablo link-time rewriting framework [4, 5] that has previously been used for obfuscation [6, 7] and steganography [8]. This prototype diversifier applies the following transformations. *Inlining* [9], *tail duplication* [9] and *two-way predicate insertion* [10] involve code duplication. *Identical function elimination*, *basic block factoring* and *function epilogue factoring* [11] all involve the replacement of duplicate code fragments by a single copy. All of those transformations not only generate diversity, they also break the assumption that there is a one-to-one mapping between two program versions' instructions. Thus, they can fool matchers that limit the number of accepted matches as discussed in Section 2.1. Furthermore, our prototype applies *control-flow flattening* [12], *branch indirection through branch functions* [13], and *opaque predicate insertion* [10]. These transformations originate from the field of program obfuscation. Fundamentally, they add unrealizable control paths of which it is hard to recognize their unrealizability. As such, they make the number of paths in the control flow graph explode, and thus make it much harder for control flow based matchers. Finally, our prototype implements a number of randomized compiler back-end tasks [9]: randomized instruction selection (see the next section), randomized instruction scheduling, and randomized code layout. The latter two transformations thwart matchers that rely on fixed static instruction orders.

2.3 Instruction Selection

Compilers map source code operations onto the operations supported in their intermediate representation, and then map those operations onto instructions available in the instruction set architecture (ISA) for which they generate assembly code. During that second step, they often have multiple choices available, because usually many sequences of instructions are semantically equivalent. Compilers are deterministic, however, and strive not only for semantic correctness, but also optimal performance and code size, so they typically reuse the same instructions and instruction patterns a lot. Because some instructions are more useful than others for more frequently occurring operations, some instructions will be used much more frequently than others.

This instruction selection and its resulting non-uniform instruction frequency are important for two reasons. Relying on a tool like a superoptimizer that generates all possible different, but semantically equivalent code sequences, we can replace the deterministic behavior of a compiler's code selection by a randomized selection to diversify programs, as mentioned in Section 2.2. In our prototype diversifier, we randomized the instruction selection by selectively replacing single instructions by alternative single instructions. The alternatives are taken from a list of equivalent instructions that was produced by a superoptimizer [14].

Secondly, it is important to understand that the non-uniform distribution of instruction frequencies can be exploited by a matcher. For any matcher based on instruction encoding, the infrequently occurring instruction types will be easy targets, as the matcher has to find matches among less candidates than it needs to do for frequently occurring instructions. As a defense against collusion attacks and matchers, we hence propose to remove as many of the infrequently occurring instructions as possible by applying ISL. This will not only make it harder for instruction encoding based matchers, but it will also do so for matchers based on control flow and data flow. The latter will happen when single instructions are replaced by sequences of multiple instructions, as this replacement inserts new instructions and new data flow. Because of the new instructions, limiting matchers as discussed in Section 2.1, will also be hampered.

We should note that, to some extent, ISL can undo the diversification obtained through randomized instruction selection. This effect can be limited, however, by applying the ISL to different instruction types. Consider for example, the `lea` (load effective address) instruction in the x86 ISA. This (large) general-purpose instruction combines a lot of computations, and can be used as an alternative to many, more specific, shorter instructions during instruction selection randomization. As the `lea` instruction occurs frequently, however, it is not a good candidate for instruction set elimination.

3 Instruction Set Limitation

Figure 1 depicts a histogram for instructions occurring in the bzip2 benchmark. Some of the infrequently occurring instructions cannot easily be replaced by other instructions, such as the x86 instructions `hlt`, `cpuid`, `in`, `int`, `iret`, `lmsw`, `out`, and `smsw`. These instructions have very specific semantics for doing IO, for communicating with the operating system, and for accessing special processor components. Other instructions, however, can easily be replaced by alternative instruction sequences that contain only more frequently occurring instructions.

In theory, almost all of the potential candidates can be replaced by more frequently occurring instructions. The URISC computer' ISA consists of only one instruction (subtract and branch conditionally) which corresponds to two instructions on the x86. However, replacing all instructions is not practically feasible, as it would result in unacceptably slow and large programs. As an example, just imagine how slow a multiplication implemented by means of subtracts and conditional jumps would be.

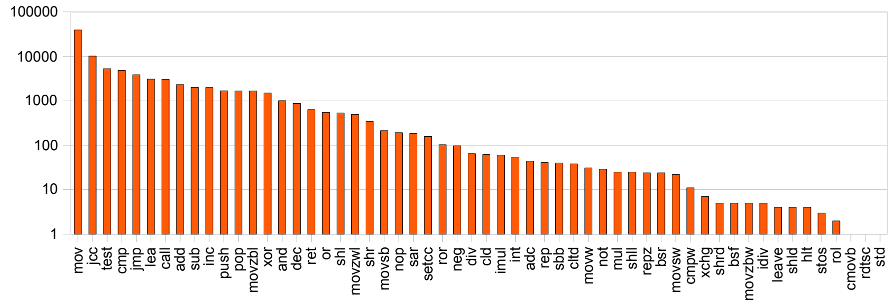

Fig. 1. Number of occurrences for each instruction in the bzip2 benchmark.

Instruction	Condition	Replacement
add	overflow and carry flags are dead	sub, (sub, push, mov, pop)
call	direct call	push, jmp
cmovcc		jcc, mov, (mov)
dec/inc	carry flag is dead	sub/add
jcc		jcc, (jmp)
leave		mov, pop
neg		sub, mov, mov
pop/push	flags are dead	mov, add/sub
ret	free register available	pop, jmp
sbb		jcc, sub, sub
setcc		jcc, mov, mov
test		and
xchg	program is single-threaded	mov, mov, mov

Table 2. Candidate instructions for limitation. Instructions between brackets denote instructions that are not needed in all cases.

On the other hand, ISL should not be limited to infrequently occurring instructions. Consider the `test` instruction in Figure 1. This instruction occurs about five times more frequently than the `and` instruction. Still it makes sense to replace the `test` instruction by `and` instructions. Because all `test` instructions can be replaced with the `and` instruction, the final result will be that there will be six times more `and` instructions in the program, but not a single `test` instruction anymore. So even by replacing frequently occurring instructions, better distributions can be obtained to defend against matching tools.

For these reasons, we have selected the 16 instructions from Table 2 as candidates for ISL. Their replacements are shown, and the conditions in which the limitations can be applied. Some limitations can only be applied if condition flags are dead. This is the case for instructions of which the replacement sets more flags than the instruction that is replaced. Since the `xchg` instruction is used for atomic read-update-write memory accesses, it cannot be replaced by a sequence of `mov` instruction in multithreaded programs. On top of the instructions used in the replacement, additional instructions might be inserted to spill registers, i.e to free registers that are needed in the replacement code.

4 An Algorithm

Let us define the quality Q of the intruction type distribution of a program p as the sum of squares of the instruction occurrence frequencies:

$$Q(p) = \sum_{\text{instruction types } i} f(i,p)\ f(i,p). \tag{1}$$

in which $f(i,p)$ denotes the number of times an instruction type i occurs in program p.

Then consider the simple case where we want to replace x instructions of type i by x instructions of type j. This will be profitable for $Q(p)$ if $x > f(i,p) - f(j,p)$. Since x is by definition positive, this condition is always true if there are less instructions of type i than of type j. Otherwise, x has to be high enough to improve the quality of the type distribution.

Let us further define the cost of a program as the number of executed instruction is a program. This number can be obtained by profiling a program to collect basic block execution counts. While the number of executed instructions is usually not a correct measure of program execution time, it is good enough for our purpose, and it reduces the complexity of the optimization problem we are facing considerably.[1] This problem consists of optimizing the quality of the instruction type distribution given a cost budget, i.e. a maximal allowed increase in number of executed instructions. Replacing a single instruction I by a sequence of k other instructions involves a cost of $(k-1) * e(I)$, in which $e(I)$ is the execution frequency of the replaced instruction, which will also be the execution frequency of the replacements. Please note that k does not only depend in the type of I but also on its context, as in some cases it might be necessary to insert spill code to free registers or condition flags. Please also note that we will only consider instructions I of which $e(I) > 0$ as observed in the profiling runs. This follows from the fact that attackers are only interested in code that is actually executed, and hence we as defenders should also only take those instructions into account.

The algorithm we propose to solve our optimization problem works as follows. It is an iterative approach in which each iteration consists of 4 steps:

1. For each instruction type i, sort all its instructions in the program and their possible replacements in ascending order of replacement cost. Instructions I with $e(i) = 0$ are not considered at all.
2. Per instruction type i, compute the smallest set of instructions for which replacing the whole set results in a positive gain ΔQ in distribution type quality. Per type i, this set is built greedily by first considering the singleton set of the first instruction in the ordered list of step 1, and by adding the next instruction from that ordered list until the set becomes large enough to have a positive effect ΔQ on $Q(p)$ when all instructions in the set would be replaced.

[1] Modeling execution time correctly for measuring the effects of static code transformations is practically infeasible.

3. From all such sets for all instruction types i, exclude sets of which the total replacement cost $Cost$, i.e. the summation of all the replacement costs of all instructions in that set, would result in exceeding the global cost budget (taking the cost of already performed replacements into account).
4. From all remaining sets, take the one with the highest fraction of gain over cost $\frac{\Delta Q}{Cost}$, and replace all instruction in that set. If there is no remaining set, the ISL terminates, otherwise it continues with step 1.

The reason why we only consider replacement cost in step 1, and not the gain in distribution quality is that the gain depends on the order in which replacements are made, while cost does not. Computing the gains correctly for all possible replacement orders is too expensive and not worthwhile.

This algorithm may apply replacements that only have a positive ΔQ because a single instruction is replaced by multiple instructions. Since such replacements will always have a higher cost than replacements that do not increase the number of instructions, this is not problematic. Cheeper replacements will be chosen first if they are available.

5 Experimental Evaluation

To evaluate the strength of our proposed instruction set elimination as a technique to fool matching tools, we performed the following experiment. For each of five SPECint20006 benchmarks, we generated two versions A and B. On each of them, we applied instruction set limitation with cost budgets of 0%, 10%, 20% and 50%, generating binaries $A_0, A_{10}, A_{20}, A_{50}, B_0, B_{10}, B_{20}, B_{50}$. With a 50% budget, the instruction set limitation is allowed to increase the estimated number of instructions (obtained through profiling) with 50%. For each of these pairs A_i and B_i, we measured their code size growth compared to A and B, their execution time increase, and their increase in distribution quality as defined by equation 1. Furthermore, for all of the program versions pairs, we measured the false positive rates and false negative rates obtained with the matching system presented in Table 1. The results are depicted in figures 2 to 7.

As can be seen from Figure 2 a cost budget of 10% already allowed to perform almost all possible instruction replacements. Only the `sjeng` benchmark required a higher budget to perform all possible replacements that help in instruction set limitation. The resulting code size increases as depicted in Figure 3 have very similar graph shapes, albeit with slightly lower numerical values. So on average, the replacement are slightly less than twice as big as the instructions they replace.

The fact that the curves in Figure 4 are monotonically increasing when big slowdowns are seen (as for `libquantum` and `sjeng`) learns us that the cost used in Section 4 can be used to control the slowdown. And for all benchmarks but `sjeng`, the performance penalty is lower than or equal to the budget used. However, this is more due to lack of replacement opportunities than to the accuracy of our cost function. Indeed, `sjeng` shows that our cost function is not an accurate predication of the actual slowdown. New research for better cost functions

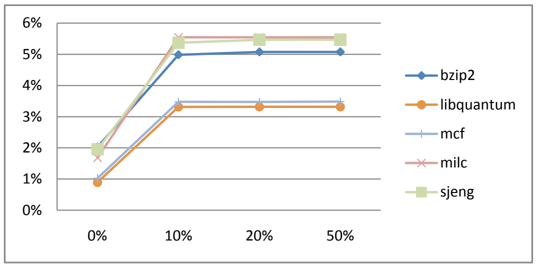

Fig. 2. The fraction of replaced instructions per cost budget.

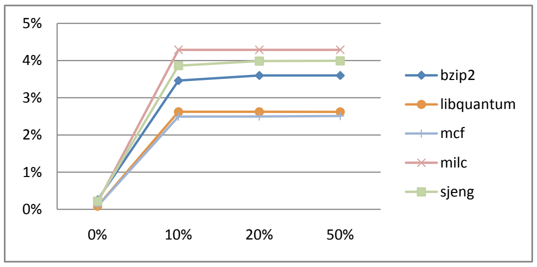

Fig. 3. The resulting code size increase.

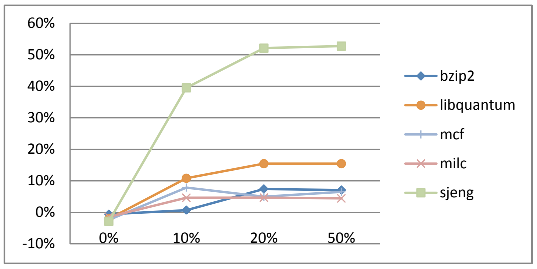

Fig. 4. Slowdown per cost budget

is hence definitely needed. More research is also needed to understand the slight performance improvement witnessed for the 0% budget. We believe this to be a side-effect of the complex interaction between the different diversifying transformations applied by our tool and the ISL, but so far we have not been able to find the exact reason.[2]

The influence on the matching capabilities of our matching system described in Table 1 is depicted in Figures 5 and 6. First, in can be observed that the false negative rates increase significantly. This means that the matcher finds far fewer corresponding instruction pairs in the two diversified program versions.

[2] Using performance counters, we were able to rule out accidental changes in branch predictor performance and cache performances.

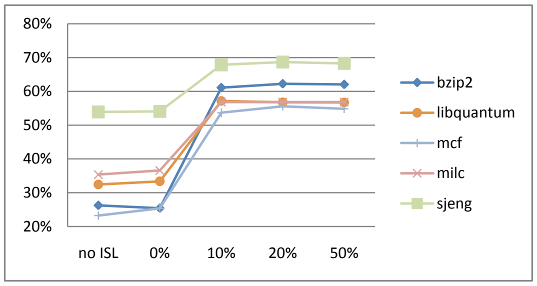

Fig. 5. False-negative rates of our matcher.

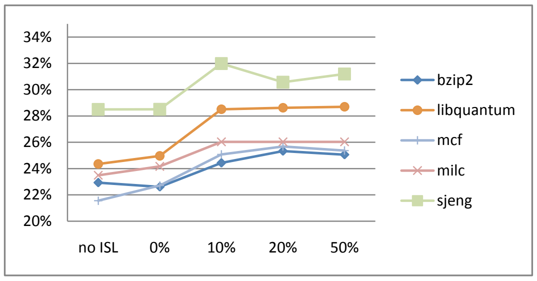

Fig. 6. False-positive rates of our matcher.

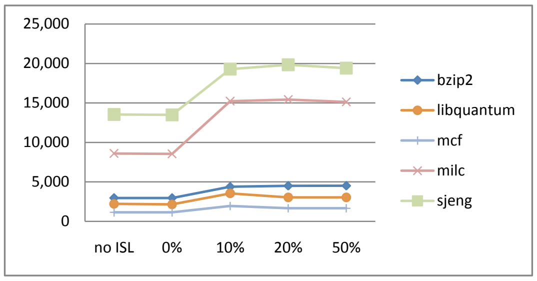

Fig. 7. Computation times (in seconds) required by the matcher

The false negative rates after ISL are roughly between 55% and 70%, while they were between about 25% and 35% before ISL, with the exception of `sjeng`, which already was at 54%. Thus, we can conclude that ISL indeed makes it harder for a matcher to find corresponding instructions in diversified program versions. To some extent, this is the result of our matchers in iterations 2, 3, 9, 11, 12, 14, 15, 18 and 21, which rely on instruction types. However, without the these matchers, the false rates would have been worse when no ISL was applied at all. Furthermore, the small increment when using a 0% cost budget indicates that ISL only helps when all, or close to all, replaceable instructions are replaced.

The false positive ratios increase much less than the false negative ratios. The reason is that our matcher is rather conservative, and will not likely match instructions of different types. Given that we only change a small fraction of the instructions in a program, ISL will not make the matcher match much more instruction. The fact that the false positive rates are not monotonically increasing results from the complex and unpredictable interaction between the different iterations in our matching system.

Finally, it is clear from Figure 7 that ISL not only thwarts the matcher to the extent that it produces much higher false results, it also requires the matcher to perform much more computations. As a result, it requires up to 72 % more time to execute our matcher (programmed in non-optimized C#, and executed on a 2.8 GHz P4) after ISL has been applied. This is due to the fact that the sets of instructions that are compared to each other in the different iterations, are considerably larger when ISL has been applied. Thus, an attacker not only gets less useful results from his matching tool, he also needs to wait for them longer.

6 Future Work And Conclusions

This paper proposed instruction set limitation. By itself, this is not a strong software protection technique, but when it is used in combination with software diversification, our experiments have shown that instruction set limitation succeeds in making it more difficult for an automated matching system to find corresponding code fragments in diversified software versions. This thwarting happens at acceptable levels of performance overhead.

Future work includes developing specific attacks against instruction set limitation, and finding techniques to limit instruction sequences rather than individual instructions. The latter will make it much harder to develop effective attacks.

References

1. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. (2007) 89–100
2. Maebe, J., Ronsse, M., De Bosschere, K.: DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In: Compendium of Workshops and Tutorials in Conjunction with the 11th International Conference on Parallel Architectures and Compilation Techniques. (2002) count 11
3. Anckaert, B.: Diversity for Software Protection. PhD thesis, Ghent University (2008)
4. De Bus, B.: Reliable, Retargetable and Extensivle Link-Time Program Rewriting. PhD thesis, Ghent University (2005)
5. De Sutter, B., Van Put, L., Chanet, D., De Bus, B., De Bosschere, K.: Link-time compaction and optimization of ARM executables. Trans. on Embedded Computing Sys. **6**(1) (2007) 5

6. Madou, M., Anckaert, B., De Sutter, B., De Bosschere, K.: Hybrid static-dynamic attacks against software protection mechanisms. In: Proceedings of the 5th ACM workshop on Digital Rights Management, ACM Press (2005) 75–82
7. Madou, M., Van Put, L., De Bosschere, K.: Loco: An interactive code (de)obfuscation tool. In: Proceedings of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06). (2006) http://www.elis.ugent.be/diablo/obfuscation.
8. Anckaert, B., De Sutter, B., Chanet, D., De Bosschere, K.: Steganography for executables and code transformation signatures. In: Proceedings of the 7th International Conference on Information Security and Cryptology. Volume 3506 of Lecture Notes in Computer Science., Springer-Verlag (2005) 425–439
9. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
10. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th Conference on Principles of Programming Languages, ACM Press (1998) 184–196
11. De Sutter, B., De Bus, B., De Bosschere, K.: Sifting out the mud: low level C++ code reuse. In: OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. (2002) 275–291
12. Wang, Z., Pierce, K., McFarling, S.: Bmat – a binary matching tools for stale profile propagation. The Journal of Instruction-Level Parallelism **2** (2000) 1–20
13. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, ACM Press (2003) 290–299
14. Massalin, H.: Superoptimizer: a look at the smallest program. In: Proceedings of the 2nd International Conference on Architectual Support for Programming Languages and Operating Systems, IEEE Computer Society Press (1987) 122–126